Compilation of Structured Polyhedral Equations

Yun Zou, Guillaume Iooss, and Sanjay Rajopadhye
Computer Science Department
Colorado State University, Ft. Collins, CO
zou_yun@yahoo.com, guillaume.iooss@gmail.com, Sanjay.Rajopadhye@colostate.edu

Abstract—**The polyhedral model is an established mathematical formalism for automatic parallelization of an important class of programs. In 1989, Mauras defined ALPHA, a polyhedral equational language based on systems of affine recurrence equations over polyhedral domains. In 1995, Dupont de Dinechin introduced *subsystems* in ALPHA to allow modularity and structured programming. Such hierarchical structure is now regaining importance when polyhedral programs are tiled with parametric tile sizes. Such tiling transformations are non-linear, and render the program representation "out of the scope" of polyhedral representation. Current approaches apply them at the final, code-generation step. Structured polyhedral programs offer an alternative solution, the ability to work with a polyhedral specification of programs with parametric tile sizes. This paper addresses the issue of code generation and compilation of such structured ALPHA programs. We implement this in our polyhedral transformation framework (AlphaZ). We also give examples to illustrate the ability to express the complex programs and to demonstrate the code generation.**

I. INTRODUCTION

The *polyhedral model* is a powerful framework for automatic parallelization and program optimization. It provides a mathematical abstraction to reason about a class of imperative programs, called affine control loop programs (the inaccurate term SCoP or static control parts is popular, but incorrect—an FFT program has static but non-affine control, and cannot be analyzed by polyhedral methods). It enables *instance-wise* analysis of statements in a program (since statements may be surrounded by loops and therefore have multiple instances) and *element-wise* analysis of arrays (rather than treating all arrays as single monolithic entities). However, underlying the program model is a declarative formalism called systems of affine recurrence equations.

ALPHA [1] is a strongly typed, functional polyhedral language, originally defined by Mauras [1] in 1989, and extended by Le Verge [2] to include reductions as first class expressions. ALPHAZ [3] is a tool developed at CSU that allows the manipulation, analysis and transformation of the ALPHA programs. It is similar in structure to an earlier tool MMALPHA [4] developed at IRISA, Rennes, France. MMALPHA targeted VLSI processor arrays and generated code in the form of a hardware description language, while ALPHAZ targets modern multi- and many-core processors, and generates parallel code (C+OpenMP and C+MPI).

One key limitation of MMALPHA was that it did not handle tiling, which was then, and still is, a difficult problem. However, tiling is a critical program transformation and is used extensively to address different aspects of performance, ranging from data locality to granularity of parallelism, and control of communication-computation balance. Parametric tiling—tiling when tile sizes are parameters to be instantiated later in the compilation process, or even at run time—is a known limitation of polyhedral model.[1] Most current polyhedral tools do not handle parametric tiling cleanly. The fundamental difficulty is that tiling with parametric tile sizes is a non-linear transformation, and therefore violates the closure properties necessary for program transformations. As a result, the "workaround" to the problem is to do parametric tiling only at the final, code-generation step when there is no need for further polyhedral analyses and transformations. All the state-of-the-art polyhedral tools adopt this strategy [5], [3] or limit themselves to fixed size tiling [6], [7], [8].

However, this is not very satisfactory, since the representation of a tiled program is no longer polyhedral, and therefore cannot be subject to further polyhedral transformation or analysis. For example, the output produced by most current polyhedral compilers cannot be fed back directly into a polyhedral compilation/optimization system.

One alternative is to perform the necessary non-linear transformations *before* any polyhedral analysis and transformations, and work with a purely polyhedral representation. At the expense of an initial, possibly human-guided, and non-obvious preprocessing, one has a polyhedral representation throughout the transformation and code generation process. This motivates the need for tools and systems to manipulate *hierarchical* polyhedral programs—polyhedral programs in which tiling has been performed at the source/algorithmic level.

In 1995, Dupont de Dinechin [9], [10] argued for the need to include modularity in ALPHA, and proposed the notion of *subsystem*s in the language. He provided a semantics for such programs and presented a number of basic analyses and transformations. The use of subsystems was limited to structural description of regular hardware in MMALPHA, notably in the AlpHard language [11]. The need for tiling, and in particular, the idea of performing tiling as a first, non-linear preprocessing transformation, motivates us to revisit subsystems in ALPHA, and is the topic of the current paper. Our main contributions are as follows:

- We revisit the definition and foundations of subsystems

[1]When tile sizes a compile-time constants, tiling can be described as an affine transformation and can be handled within the polyhedral model.

Expression	Syntax	Domain	Comment
Constant	Constant symbol	$\mathbb{Z}^0$	
Variable	Variable name X	$\mathcal{D}_X$	Declared domain
Operator	`op(`Exp_1`, ...,` Exp_M`)`	$\bigcap_{i=1}^{M} \mathcal{D}_{Exp_i}$	infix syntax too
Case	`case` Exp_1`; ...;` Exp_M `esac`	$\biguplus_{i=1}^{M} \mathcal{D}_{Exp_i}$	disjoint union
Restriction	$\mathcal{D}$ `:` Exp	$\mathcal{D} \cap \mathcal{D}_{Exp}$	relational inverse
Dependence	$f@Exp$	$f^{-1}(\mathcal{D}_{Exp})$	
Reduction	`reduce(`$\oplus$`,` f`,` $Expr$`)`	$f(\mathcal{D}_{Expr})$	

Fig. 1. ALPHA expressions and computation rules for the associated domains. A *case expression* describes a conditional expression defined over disjoint domains. A *restriction expression* just restricts the definition domain of its sub-expression. The value of a *dependence expression* $f@Expr$ at an index point $\vec{i}$ is the value of $Expr$ at the point $f(\vec{i})$. In the case where the sub-expression is a variable X, the value of $f@X$ at the point $\vec{i}$ can be written as either $(f@X)[\vec{i}]$ or $X[f(\vec{i})]$. A *reduction expression* corresponds to an accumulation with the operator $\oplus$. The values of $Expr$ at all points that are mapped by f to the same index point i are accumulated together, using the associative and commutative operator $\oplus$.

in ALPHA as proposed by Dupont de Dinechin [9], [10] and propose new static analyses and transformations. In particular, we extend a critical program transformation called Change-of-Basis to operate on ALPHA programs with subsystems. We also implement in ALPHAZ, some known transformations and analysis that were not previously implemented in MMALPHA.

- We describe a methodology for compiling ALPHA programs with subsystems to parallel, shared memory code (sequential code is a special case) in OpenMP, and implement a code generator embodying this methodology. Besides the correctness of the generated code, our code generator is able to generate code incorporating a number of optimizations such as memory reuse and value reuse.

II. BACKGROUND

In this section, we briefly introduce ALPHA[2], including Dupont de Dinechin's extension to subsystems.

A. The ALPHA Language

An ALPHA program is a collection of *affine systems*, each affine system consists of:

- A *parameter domain*, which is a polyhedral domain that specifies the constraints on the program parameters.
- A list of declarations of input/output/local *variables* `Var`, each associated with a polyhedral domain $\mathcal{D}_{\texttt{Var}}$.
- A list of equations `Var = Expr`. There is exactly one equation for each local or output variable. The syntax of expressions is given in Fig 1.

The analysis phase computes two domains for each sub-expression:

- *Expression domain*, which is the set of points in which the expression is defined.
- *Context domain*, which is a subset of the expression domain, and is the set of points at which the expression needs to be evaluated, in order to compute the outputs of the system.

The expression domain is computed recursively, in a bottom-up fashion, using the rules in the last column of Fig 1. The context domain is computed in a top-down fashion.

Example: Consider the equation $A.x = b$ where A is a lower triangular matrix whose diagonal values are 1 and x, b are vectors. Given A and b, forward substitution solves this equation for x using the formula, $x_i = b_i - \sum_{j=0}^{i-1} A_{i,j}x_j$, with appropriate boundary cases. An ALPHA program is shown in Fig 2.

B. Subsystems

Dupont de Dinechin [9], [10] introduced the notion of modularity in ALPHA. A *subsystem* is an affine system which is "called" or invoked from another system on a specific set of parameters and inputs values. Indeed, a "polyhedral collection" of systems may be invoked in a compact manner. Fig 3 shows a modular version of the forward substitution example above, where each instance of the reduction is written as the invocation of a dot-product subsystem.

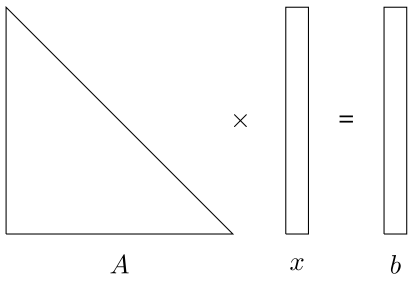
The equation describing this system call is called a *use equation* and has the following syntax:

`use` $\mathcal{D}_{ext}$ `s[p] (i) (o);`

where:

- $\mathcal{D}_{ext}$ is the [optional] *extension domain*.
- `s` is the name of the subsystem called.

[2]A detailed presentation of the formalism can be found in [12]



```
affine FS {N | N>0 } // System name & parameter domain
input                // List of input variables
   float A {i,j | 0<=j<i<N};
   float b { i  | 0<=i<N };
output               // List of output variables
   float x { i  | 0<=i<N };
let                  // List of equations
   x[i] =
     case
     {i| i == 0}: b[i];
     {i| i > 0 }: b[i] - reduce(+, (i->), A[i,j]*x[j]);
     esac;
.
```

Fig. 2. Forward substitution in ALPHA: a program to solve $Ax = b$ for a unit lower triangular matrix, A.

```
affine FSwithDotProd {N| N> 0}
input
   float A {i,j | 0<=j<i<N};
   float b {i | 0<=i<N};
output
   float x {i | 0<=i<N};
local
   float temp {i | 1<=i<N};
let
   use {i|1<=i<N}   // instantiate a 1-D set, indexed by i
       dotProd[i-1] // of dotProd, where the i-th instance
                    // has size parameter i-1.
      // with the following input and output arguments:
      (A, (i,j->j)@x) returns (temp);
   x = case
            {i| i == 0}: b;
            {i| i > 0 }: b - temp;
         esac;
.
```

Fig. 3. Forward substitution in ALPHA with subsystems

- p is a list of affine index expressions specifying size parameters of the instances.
- i is a list of the *input expressions*.
- o is a list of the *output variables*.

Let us first consider the particular case where the extension is not specified in the use equation. This equation is calling the affine system s with the parameter values p and the inputs i. The outputs of the subsystem define the values of the variable o. And the index expressions of p are affine expressions of the parameters. In addition, because no extension domain was specified, we only have a single call to the subsystem.

The extension domain allow us to express a parametric number of calls to the subsystem s, each points $\vec{i}_{Ext}$ of the extension domain corresponding to an instance of subsystem call. For the $\vec{}_{Ext}$th call, the input values sent are the values of the input expressions i, whose first few dimensions are set to $\vec{i}_{Ext}$. The output values produced by this subsystem call define the values of o whose few first dimensions are $\vec{i}_{Ext}$. It is also possible to give distinct parameter values to each subsystem call using index names of $\vec{i}_{Ext}$.

III. Static Analysis of Subsystems

We first present our adaptation of the context domain computation to subsystems. Then we describe the Change-of-basis transformation on use equations and explain how to extend a data structure called the polyhedral reduced dependence graph (PRDG) to use equations.

A. Context domain for the input expressions of a use equation

For a standard ALPHA equation Var = Expr, the context domain of Expr is exactly the definition domain of Var. Then, we recursively compute the context domain of each subexpression of Expr. However, the input expressions of a use equation are not included inside any standard ALPHA equation, and therefore we need to introduce new rules to handle this.

Dupont de Dinechin introduced an inlining transformation for subsystem [9, Section 4.5]. This inlining transformation creates a new local variable in the main system for each input/output variable of the inlined subsystem, and all the equations in the subsystem is copied into the main system with these new variables. Moreover, every domain or function from the subsystem is modified, to account for the extension domain and the parameter changes. Also, a copy equation is added for every input of the subsystem, and has the form $\texttt{Var}_{in}$= Expr, for the input equations.

To determine the set of points of an input expression which will be used by a subsystem, we can translate the domain of the corresponding input variable in the subsystem, using the transformation introduced by Dupont de Dinechin for his inlining transformation. This is equivalent to consider the corresponding input copy equation, then apply the context domain computation on this standard equation.

B. Change of Basis on a use equation

The Change of Basis (CoB) is a very useful transformation which subsumes many others (such as loop skewing, interchanging) in a single formalism. It consists of transforming the domain $\mathcal{D}_{Var}$ of a variable into $f(\mathcal{D}_{Var})$, where f is a bijective affine function. This transformation impacts the affine system in the following way.

- The equation Var = Expr is transformed into $\texttt{Var} = \texttt{f}^{-1}\texttt{@Expr}$
- Each occurrence of Var is replaced by f@Var

The CoB transformation cannot be applied directly to an output variable $\texttt{Var}_{out}$ of a use equation. Indeed, if f touches the dimensions of the extension domain, it modifies the domains of all the outputs of the use equation. The ALPHAZ system returns an error and advices the user to introduce a copy variable. However, the user can force the transformation, in this case, we perform the same CoB to every output variable of the use equation.

It is also possible to apply a CoB directly on a use equation: the idea is to transform its extension domain $\mathcal{D}_{ext}$ into $f(\mathcal{D}_{ext})$ where f is bijective. This transformation will impact the affine system in the following ways:

- The extension domain $\mathcal{D}_{ext}$ is replaced by $f(\mathcal{D}_{ext})$
- Each input expression $Expr$ is replaced by $(f^{-1} \times Id_k)@Expr$ where k is the dimension of the input of the subsystem.
- We perform partially a CoB on each output variable, using the function $(f \times Id_{k'})$ where k' is the dimension of the output of the subsystem.

This transformation is used to permutate or skew a loop nest, to enable other transformations (such as tiling) or to expose parallelism.

C. Outlining transformation

The outlining transformation consists of extracting a set of equations from a main system, and creating a subsystem from them. The input variables of this new subsystem are the variables used in these equations and defined elsewhere. The output variables are the variables defined in these equations and used elsewhere. The local variables are the variables defined in these equations and used only inside these equations.

Our current version is simple: the use equation do not have any extension domain, and input expressions to the use equation can be specified by the user. Eventually, we will allow the user to define an extension domain, which will impact the domain of the variables of the subsystem and the use equation created.

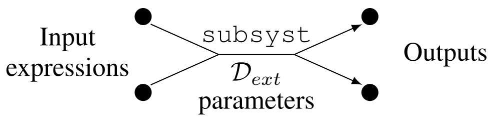
D. Polyhedral Reduced Dependence Graph

To analyze statically the dependencies of a program, we have to create a *Polyhedral Reduced Dependence Graph* (PRDG). The nodes in a PRDG are the variables of a program. There is an edge between from node a (source) to node b (destination) if the computation for variable a depends on variable b. The nodes and edges are labeled respectively by the domain of the variable and the dependence function.

We chose to represent a use equation in a PRDG in the following way:

- A new node is created for each input expression
- The use equation is represented by an hyper-edge, whose sources are the previously created nodes for input expressions, and destinations are the nodes for the output variables.

Moreover, we label the hyper-edge by the name of the subsystem called, the parameters and the extension domain of the use equation.



IV. CODE GENERATION

Code generation is an important component in ALPHAZ, and several code generators are available [3]. The most sophisticated one *ScheduledC*, generates C+OpenMP code. We extended this code generator to support code generation for subsystems. In this section, we will describe the strategy for the code generator and illustrate how to use it to generate efficient code.

In ALPHAZ, compilation is based on three orthogonal components.

- First, a *target mapping* specifies, for each variable in the ALPHA program, all the aspects of the desired parallelization. This includes the schedule, the memory allocation, and tiling—which dimensions to tile, how many levels of tiling, and the tile sizes at each level.
- Next, a verifier checks the legality of the proposed mapping (this is because the target mapping may optionally be specified interactively by the user).
- Finally, a code generator in the form of a very sophisticated "pretty printer" based on CLooG [13] that actually produces the target code.

The choice of the target mapping is a long standing research problem, since the target architectures are continually evolving. Therefore ALPHAZ provides the ability for users to explore these choices manually.

A. Target Mapping

The main components of the target mapping are *space-time map* and *memory map*. The *space-time map* has two components, a schedule and a processor allocation, thus it determines when and on which virtual processor the computation should be executed. A space-time map is specified for each variable, and is a bijective multi-dimensional affine function [14], each dimension can be sequential, parallel or tagged as an "ordering" dimension. For example, we can give a space-time map $(i \rightarrow i, 0)$ to the equation x in the forward substitution example. This tells the code generator that the statement generated for the ith index of x is to be computed at time step $(i, 0)$, and the second dimension is an ordering dimension. The *memory map*, represents, for every variable in the program, its memory allocation, and is also specified by a multi-dimensional affine function, possibly annotated with modulo functions. For example, we can set the memory map for the variable `temp` in the forward substitution to be $(i, j \rightarrow i)$, which means the $\langle i, j \rangle$th value is stored at the memory location i.

To generate codes with parallelized C loops, the target mapping also needs to have additional information. The parallelization specification tells the code generator which loops are to be annotated with `pragma omp parallel`. Since specific loops may not correspond directy to the dimensions of an index, and so, this is specified as the nth dimension with a certain ordering prefix in the space-time map. For example, given the space-time map $(i \rightarrow i, 0)$ for statement x, we can parallelize the i dimension with the specification: the first dimension with empty ordering prefix.

In compiling ALPHA programs with subsystems, the main assumption is that instances of subsystems are atomic, i.e., they can be executed with the following protocol: first evaluate (and collect) all the inputs to (an instance of) the subsystem, then evaluate all the local and output variables of that instance, and finally, retrieve the outputs and continue execution of other computations in the main system. Atomicity can be verified by a simple extensions to the algorithm that determines the subsystem schedules [15].

B. Strategy

We now describe our strategy for subsystem code generation, and the necessary extensions to target mapping.

Because of atomicity of subsystems, we can implement them as function calls in C. Where the function call statement is placed is decided by the space-time maps in target mapping. A space-time map is specified for every use equation and its left hand side has the same number of dimensions as $\mathcal{D}_{ext}$.

In order to implement function calls, the key problem is how to pass the correct values as inputs or store the outputs into the correct place. Simply passing some pointer of the existing variable is incorrect for two reasons. First, the inputs of subsystems can be an expression and not necessarily a variable, so there may not be any memory allocated for it in the calling system. Second, the set of indices corresponding to an instance of a subsystem may be an arbitrary "slice" of the context domain of the expression, and hence, the appropriate set of values needs to be marshaled together into a contiguous region of memory.

Figure 4 shows the matrix multiplication using a `dot_product` subsystem: the $\langle i, j \rangle$th value of the final matrix C is computed by a dot product of the ith row of matrix A and the jth column of matrix B by the $\langle i, j \rangle$th instance of the use. Although tiling is not addressed in this example, it is a simple example to illustrate the code generation strategy and possible optimizations. Later, we will show more examples where tiling is used.

For this program, if both matrix A and B are allocated in the standard row-major manner, the ith row of A can be passed as `&A[i]`, but the columns of B are not in contiguous memory locations, but that is how a dot-product function expects its arguments. To generate correct codes, a temporary variable is created for every input/output of each use equation, that has the same memory size and allocation as the *declared* domains of the subsystem input/output, and corresponding values are copied to/from these variables before/after the function call.

To achieve this, our code generator generates three auxiliary statements for each input/output of each use equation: a memory allocation statement for each temporary variable, a set of value copy statements for the inputs and outputs, and the memory free statements for the temporary variables.

```
// Product of rectangular matrices A and B
affine matrix_product_SubSyst
              {N,K,M | N>0 && K>0 && M > 0}
input
    float A {i,k | 0<=i<N && 0<=k<K};
    float B {k,j | 0<=k<K && 0<=j<M};
output
    float C {i,j | 0<=i<N && 0<=j<M};
let
use {iP,jP|0<=iP<N && 0<=jP<M}
    dot_product[K]
    ((pi,pj,k->pi,k)@A,
     (pi,pj,k->k,pj)@B)
    returns (C);
.
```

Fig. 4. Matrix multiplication using dot product subsystem

```
//declare temporary variables
long* UseEquation_C_input_0;
long* UseEquation_C_input_1;
long UseEquation_C_output_0;

#pragma omp parallel for private(j)
for(i = 0; i <= N-1; i++){
  #pragma omp parallel for
  for(j = 0; j <= M-1; j++){
  //allocate memory for the inputs
  UseEquation_C_input_0 =
            memory_allocation_for_UseEquation_C_input_0(K);
  UseEquation_C_input_1 =
            memory_allocation_for_UseEquation_C_input_1(K);

  //value copy statements for inputs
  value_copy_for_UseEquation_C_input_0(K, i, j, A,
                                        UseEquation_C_input_0);
  value_copy_for_UseEquation_C_input_1(K, i, j, B,
                                        UseEquation_C_input_1);

  //function call
  dot_product(K,UseEquation_C_input_0, UseEquation_C_input_1,
                                       &UseEquation_C_output_0);

  //value copy statements for outputs
  value_copy_for_UseEquation_C_output_0(K, i, j, C,
                                       &UseEquation_C_output_0);

  free(UseEquation_C_input_0);
  free(UseEquation_C_input_1);
  }
}
```

Fig. 5. The generated code for the matrix multiplication

Figure 5 shows the code that is generated for the system *matrix_product_SubSyst*. Both of the loops in this example have been specified as parallel. Since parallelism ijut a simple annotation as far as the code generator is concerned, the remaining examples focus on sequential code.

By default, the temporary variables and special statements are created for every instance of the subsystem call, since the memory size and values that are needed for every subsystem instance may be different (e.g. in the forward substitution example of Figure 3, the i-th instance of the dotproduct system operates on vectors of length $i - 1$). However, in other cases, reuse may be possible. For example, in the matrix

```
//declare temporary variable
long* UseEquation_C_input_0;
long* UseEquation_C_input_1;
long UseEquation_C_output_0;

//allocate memory for the inputs
UseEquation_C_input_0 = memory_allocation_for_UseEquation_C_input_0(K);
UseEquation_C_input_1 = memory_allocation_for_UseEquation_C_input_1(K);
for(i = 0; i <= N-1; i++){
value_copy_for_UseEquation_C_input_0(K,i,j,A,UseEquation_C_input_0);
    for(j = 0; j <= M-1; j++){
        //value copy statements for inputs
        value_copy_for_UseEquation_C_input_1(K,i,j,B,UseEquation_C_input_1);

        //function call
        dot_product(K,UseEquation_C_input_0, UseEquation_C_input_1,
                    &UseEquation_C_output_0);

        //value copy statements for outputs
        value_copy_for_UseEquation_C_output_0(K,i,j,C, &UseEquation_C_output_0);

    }
}

//memory free
free(UseEquation_C_input_0);
free(UseEquation_C_input_1);
```

Fig. 6. The generated code for the matrix multiplication with memory and value reuse

multiplication example, the temporary variable size for both inputs are the same across all the subsystem calls. Therefore, the same memory allocation can be shared across all the subsystem calls for each input. In other words, the memory allocation statement can be *hoisted out* of the whole loop nest for the subsystem call. Furthermore, all the jth instances of the subsystem call use the *same value* for the first input, so the value copy statement for the first input can also be hoisted out the inner j loop.

Sophisticated polyhedral analysis can be used to determine such optimizations, but as far as th code generator is concerned, they are eventually specified as space-time maps for the three auxiliary statements. For the matrix multiplication example, we set the space-time map of the allocation statements for both inputs to be $(ip, jp \rightarrow 0, 0, 0, 0, 0)$, (note that the first, third and fifth dimensions are ordering dimensions). In order to hoist this statement out of the i and j loops, we simply also set the second and fourth dimensions to zero. In general, for any statement that needs to be hoisted out k levels of a m-nested loop, we just need to set the values of its space-time map that correspond to the innermost k loop dimensions to be zero. Conversely, given a space-time map of an auxiliary statement, we can figure out the value k by counting the number of successive innermost loops with a zero value.

For the matrix multiplication example, we can completely hoist the memory allocation statements for both inputs, and the value copy statement for the first input by one level (the copy of the i-th row of A is reused across the inner j loop). The code generated with these three optimizations is shown in Figure 6.

C. Legality check for target mapping

Our code generator accepts the target mapping as gospel and generates code that respects the given mapping. A separate module in the system There is responsible for ensuring that the proposed target mapping is legal. For code generated without subsystems, this legality check is performed using existing polyhedral analyses [16]. We now explain how these need to be extended to handle the legality of target mappings for subsystems. Due to the space limitations, we only describe the legality checks for the sequential case.

In addition to the legality of the susbystem schedule including the test of atomicity, two additional checks need to be performed for the target mapping validation for subsystems: *dependency check* and *reuse check*. The *dependency check* validates whether the space-time map correctly specifies the loop hoisting optimization and whether all the dependencies in the program are respected. The *reuse check* checks whether the memory allocation and copy respect the lifetimes of the values concerned. The dependency check has the following components:

1) By default, the special statements for the inputs/outputs of a subsystem are placed within the same loop with the subsystem call (as shown in Figure 5). This requires the space-time map for the special statements to have the same linear part as the space-time map for the use equation. Assume the space-time map for a special statement is Θ_s, the schedule for the use equation is Θ_{use}, and $lin(\Theta)$ represents the linear part of the space-time map, then we have to check $lin(\Theta_s) = lin(\Theta_{use})$. When a special statement is hoisted out, the special statement should still share a common loop prefix with the subsystem call. Assume that the statement is hoisted out by k levels, and there are m loops. Then we have to check $k \leq m$, and that the first $(m-k)$ dimensions of $lin(\Theta_s)$are the same as the $lin(\Theta_{use})$, and the remaining k dimensions are set to zero.
2) The dependencies between the special statements and the subsystem call have to be respected (these dependencies are in addition to those that were in the original program). For example, the memory has to be allocated before any value is assigned, therefore, the value copy statement must happen after the malloc statement. Let $\Theta^{malloc}_{input_i}$, $\Theta^{copy}_{input_i}$, $\Theta^{free}_{input_i}$ be the schedule for the memory allocation statement, value copy statement and memory free statement of the ith input. Similarly, we also have $\Theta^{malloc}_{output_i}$, $\Theta^{copy}_{output_i}$, $\Theta^{free}_{output_i}$ for the ith output. Then the check we have to do is to ensure that $\forall z \in \mathcal{D}_{ext}$,
$$\Theta^{malloc}_{input_i}(z) \prec \Theta^{copy}_{input_i}(z) \prec \Theta_{use}(z) \prec \Theta^{free}_{input_i}(z)$$
$$\Theta^{malloc}_{output_i}(z) \prec \Theta^{copy}_{output_i}(z) \prec \Theta^{free}_{output_i}(z)$$
$$\Theta_{use}(z) \prec \Theta^{copy}_{output_i}(z)$$
3) All other dependences in the program have to be respected by the space-time maps, and this is exactly what the ALPHAZ verifier already does for programs without subsystems [16]. In adition, the space-time map must be checked for atomicity of subsystems.

The *reuse check* checks whether the memory reuse and value reuse is legal. The zth instance of the subsystem call can share the same memory allocation statement with the z'th instance of the subsystem call for the ith input/output, if and only if the memory size for the ith input is the same for the two instances of the subsystem. When the memory allocation statement is hoisted out the m-nested loops for k levels, the subsystem calls with the common first $(m-k)$ indices all must have the same memory size for the ith input. Therefore, the vertices of domain for the ith input of the subsystem should not depend on any of the inner k loop indices. This is a simple check using existing polyhedral machinery.

For the value reuse, the zth instance of the subsystem call can share the same value copy statement with the z'th instance of the subsystem call for the ith input, if and only if the values

```
affine BlockMM {n, b| (n,b)>0}
given
    float A, B {ii,jj,i,j |
            0<= (ii,jj)<n && 0<=(i,j)<b};
returns
    float C {ii,jj,i,j |
            0<= (ii,jj)<n && 0<=(i,j)<b};
using
    float CC {ii,jj,kk,i,j |
          0<= (ii,jj,kk)<n && 0<=(i,j)<b};
through
    use {ii, jj, kk | 0<=(ii,jj,kk)<n}
        MatrixMult[b]
        ((ii,jj,kk,i,j->ii,kk,i,j)@A,
        (ii,jj,kk,i,j->kk,jj,i,j)@B)
        returns (CC);
    C = reduce(+,
          (ii,jj,kk,i,j->ii,jj,i,j), CC);
.
```

Fig. 7. The ALPHA program for tiled matrix multiplication

needed by both instances are the same. A value copy statement can be hoisted out the innermost k loops if it has a reuse along the direction defined by $[0, 1_k]$, the vector which is 1 in the innermost k dimensions and 0 in the outer ones. Again, we can use known polyhedral machinery [17] to perform these checks.

V. RESULTS

So far, we used simple examples to illustrate subsystems and our compilation strategy. Now, we demonstrate how to write structured *tiled* ALPHA code for the tiled forward substitution and tiled matrix multiplication, and also show the sample codes we generated with different optimization.

A. Tiled Matrix Multiplication

1) Alpha program for tiled matrix multiplication: In matrix multiplication, tiling can be applied to explore a better data locality when the size of the matrix is too large to fit into the cache. Such a tiled program is shown in Figure 7, whose matrix size and block size are squares.

2) Generated code for tiled matrix multiplication: For the tiled matrix multiplication program, the memory size used for each input/output for all subsystem calls are the same. Therefore, all the subsystem calls can share the same memory allocation for each input/output, i.e., the malloc statements can be hoisted out all the way. Also, the accumulation for the final answer C can happen right after each of the kkth blocks is computed—the reduction is interspersed with the function calls. The generated with all these optimization is shown in Figure 8.

B. Tiled Forward Substitution

1) Alpha program for tiled forward substitution: The forwad substitution program can also be blocked into square tiles

```
//declare temporary variables
float** UseEquation_CC_input_0;
. . . //other temporary variables

//memory allocation for the temporary variables
UseEquation_CC_input_0 =
      memory_allocation_for_UseEquation_CC_input_0(n,b,...);
. . .   //memory allocation for other temporary variables
for(i=0;i <= n-1;i+=1){
  for(j=0;j <= n-1;j+=1){
    for(k=0;k <= n-1;k+=1){
    //copy corresponding values into temp input variables
    value_copy_for_UseEquation_CC_input_0(n,b,...);
    value_copy_for_UseEquation_CC_input_1(n,b,...);

    //subsystem call
      MatrixMult(b,UseEquation_CC_input_0,...);

    //copy corresponding values into temp output variables
    value_copy_for_UseEquation_CC_output_0(n,b,...);
    }
    //compute the answer for the <i,j>th block of C
    for(k=0;k <= n-1;k+=1){
      for(l=0;l <= n-1;l+=1){
        C(i,j) = reduction(CC,...);
      }
    }
  }
}
//memory free for the temporary variables
memory_free_for_UseEquation_CC_input_0(n,b,...);
.  .  . //memory free for other temporary variables
```

Fig. 8.   The parallelized tiled matrix multiplication

(the diagonal blocks will be triangular) tiles The corresponding program is shown in Figure 9.

2) Generated code for forward substitution: To generate efficient code for the forward substitution, we have to discover the reuse first. There are two use equations in the system *FSBlock*, one for a smalles forward substitution system, and the other for a matrix-vector product. In both use equation, the mallocs for inputs and outputs can be hoisted completely. No value reuse can be enabled for the forward substitution. The code generated is shown in Figure 10.

VI. Related Work

The polyhedral model has recently achieved a significant success in automatic parallelization.[3] It is currently used in research compilers like Loopo [18], Pluto [6] (see `http://www.cse.ohio-state.edu/~bondhugu/pluto`) and the PoCC tools of the Alchemy group in INRIA, Saclay [5]. More importantly, the polyhedral model is now included in at least three production compilers. IBM's XL compiler family now adopts this model as the intermediate representation for loops; the 4.4.0 release of `gcc` now includes this as part of the GRAPHITE project; and Reservoir Labs also uses it in their compilation engine [8].

Alpha [1] is a strongly typed functional language developed by Mauras at IRISA, and extended by Le Verge to include *reductions* as first class expressions [2], and by Dupont de Dinechin [9], [10] with modular structure. This abstraction has allowed us to develop extremely powerful analyses and transformations such as memory re-allocation [3], detection

[3]Pingali's keynote at the 2010 LCPC Workshop listed the polyhedral model as one of just three "successes" of compilers in the past 25 years.

```
affine FSBlock {N, B | (N,B) >1}
given
float LL {ii, jj, i, j | 0<=jj<ii<N && 0<= (i,j)<B};
float DD {ii, i, j | 0<=ii<N && 0<=j<=i < B};
float BB {ii, i | 0<=ii<N && 0<=i<B};
returns
float XX {ii, i | 0<=ii<N && 0<=i<B};
using
float AccVec {ii, jj, i | 0<=jj<ii<N && 0<=i<B};
float AccSum {ii, i | 0< ii<N && 0<=i<B};
float sum {ii,i | 0<= ii<N && 0<=i<B};
through
   use{ii,jj|0<=jj<ii<N} blockMVM[B] (LL,(ii,jj,j->jj,j)@XX)
                                                  returns (AccVec);
   use {ii | 0<=ii<N} diagSolve[B] (DD,BB-sum) returns (XX);
   sum[ii,i] = case
    {|ii==0}: 0;
    {|ii > 0}: AccSum[ii,i];
   esac;
   AccSum = reduce(+, (ii,jj,i -> ii,i), AccVec);
.


affine blockMVM {B| B>0}
  ... // declarations omited for brevity
through
  y[i] = reduce(+, [j], A[i,j]*x[j]);
.


affine diagSolve {B | B>0}
  ... // declarations omited for brevity
through
  X[i] = case
    {|i==0} :b[i]/L[i,i];
    {|i>0} : (b[i] - reduce(+, [j],
                            {|j<i}:L[i,j]*X[j])) / L[i,i];
    esac;
.
```

Fig. 9.   Alpha code for the tiled forward substitution

```
//declare temporary variables
float** UseEquation_AccVec_input_0;
. . . . //other temporary variables

//memory allocation for temporary variables
UseEquation_AccVec_input_0 =
memory_allocation_for_UseEquation_AccVec_input_0(N,B,ii,...);
. . . .

for(i=0;i <= B-1;i+=1){
  sum(0,i) = 0;
}
value_copy_for_UseEquation_XX_input_0(N,B,ii,...);
value_copy_for_UseEquation_XX_input_1(N,B,ii,...);
diagSolve(B,UseEquation_XX_input_0,...);
value_copy_for_UseEquation_XX_output_0(N,B,ii,...);
for(i=1; i <= N-1; i+=1){
  for(j=0; j <= i-1; j+=1){
    value_copy_for_UseEquation_AccVec_input_0(N,B,ii,...);
    value_copy_for_UseEquation_AccVec_input_1(N,B,ii,...);
    blockMVM(B,UseEquation_AccVec_input_0,...);
    value_copy_for_UseEquation_AccVec_output_0(N,B,ii,...);
  }
  for(j=0;j <= B-1; j+=1){
    AccSum(i,i3) = reduce_FSBlock_AccSum_1(N,B,i,...);
    sum(i,j) = AccSum(i,j);
  }
  value_copy_for_UseEquation_XX_input_0(N,B,ii,...);
  value_copy_for_UseEquation_XX_input_1(N,B,ii,...);
  diagSolve(B,UseEquation_XX_input_0,...);
  value_copy_for_UseEquation_XX_output_0(N,B,ii,...);
}

//memory free
memory_free_for_UseEquation_AccVec_input_0(N,B,ii,...);
. . . .
```

Fig. 10.   Generated code for the forward substitution with memory reuse

and parallelization of scans and reductions [19] using the algebraic/mathematical properties of semi-ring matrices.

After Dupont de Dinechin's original work, there was relatively little effort to exploit modularity in ALPHA, partly because the MMALPHA tool did not support automatic tiling, which was at that time an unsolved. It is only with recent work on parametric tiled code generation [20] that modularity in polyhedral equational languages becomes important.

VII. CONCLUSION

The importance of tiling as a program optimization, and the fact that it is in general, a non-linear transformation poses some unique challenges to polyhedral compilation techniques. An approach to overcoming these difficulties is to pre-process to the program so that tiling is performed initially, at the specification level. In the polyhedral equational language, ALPHA, subsystems as introduced by Dupont de Dinechin [9], [10] provide us a very natural way of describing such tiled specifications. In this paper, we first presented some new static analyses and transformations for subsystems. We also implemented a code generator that handles subsystems. It generates code with memory and value reuse. The current system requires manual specification of the *target mapping* although our ongoing work is on automating this.

REFERENCES

[1] C. Mauras, "ALPHA: un Langage Equationnel pour la Conception et la Programmation d'Architectures Paralleles Synchrones," Ph.D. dissertation, L'Universite de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[2] H. Le Verge, "Un Environnement de Transformations de Programmmes pour la Synthese d'Architectures Regulieres," Ph.D. dissertation, L'Universite de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, Oct 1992.

[3] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "AlphaZ: A System for Design Space Exploration in the Polyhedral Model," in *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, 2012.

[4] MMAlpha, "MMAlpha: a Programming Environment for Manipulating ALPHA programs," 2009, http://www.irisa.fr/cosi/ALPHA/.

[5] PoCC, "PoCC: the Polyhedral Compiler Collection," 2012, http://www.cse.ohio-state.edu/ pouchet/software/pocc/.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, ser. CC'08/ETAPS'08. Springer-Verlag, 2008, pp. 132–146.

[7] J. C. Chun Chen and M. Hall, "CHiLL: A framework for Composing High-level Loop Transformations," University of Southern California, Tech. Rep., 2008.

[8] C. Bastoul, N. Vasilache, A. Leung, B. Meister, D. Wohlford, and L. R., "Extended static control programs as a programming model for accelerators, a case study: Targetting clearspeed csx700 with the r-stream compiler," in *First Workshop on Programming Models for Emerging Architectures (PMEA)*, Raleigh, NC, Sept 2009, p. to appear.

[9] F. Dupont de Dinechin, "Systemes Structures d'Equations Recurrentes : Mise en Oeuvre dans le Langage Alpha et Applications," Ph.D. dissertation, Universite de Rennes, IRISA, Rennes, janvier 1997.

[10] F. Dupont de Dinechin, P. Quinton, and T. Risset, "Structuration of the ALPHA Language," in *Massively Parallel Programming Models*, W. Giloi, S. Jahnichen, and B. Shriver, Eds. IEEE Conmputer Society Press, 1995, pp. 18–24.

[11] P. Le Moenner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton, "Generating Regular Arithmetic Circuits with AlpHard," in *Massively Parallel Computing Systems*, May 1996.

[12] D. K. Wilde, "The ALPHA Language," Unite de recherche Inria Roquencourt et Sophia-Antipolis, Tech. Rep., 1994.

[13] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.

[14] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[15] T. Risset, F. de Dinechin, and S. Robert, "Structured Scheduling of Recurrence Equations," INRIA, Rapport de recherche RR-3282, 1997.

[16] V. Basupalli, "The alphz verifier," Master's thesis, Colorado State University, CO, USA, Dec 2011.

[17] Gautam and S. Rajopadhye, "Simplifying Reductions," *SIGPLAN Not.*, vol. 41, no. 1, pp. 30–41, Jan. 2006.

[18] M. Griebl and C. Lengauer, "The loop parallelizer LooPo," in *CPC 1996: Sixth Workshop on Compilers for Parallel Computers*, M. Gerndt, Ed., vol. 21, 1996, pp. 311–320.

[19] Y. Zou and S. Rajopadhye, "Automatic parallelization of 'inherently sequential' nested loop programs," Colorado State University, Computer Science Dept., Tech. Rep. 11-102, March 2011.

[20] D. Kim and S. Rajopadhye, "Efficient tiled loop generation: D-tiling," in *LCPC 2009: The 22nd International Workshop on Languages and Compilers for Parallel Computing*, C. Gao, Pollock and Li, Eds. Newark, DE: Springer, LNCS, October 2009, pp. 293–307.